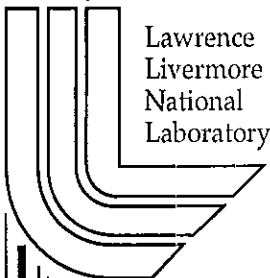


Modern Tools for Modern Software

G. Kumfert, T. Epperly

October 31, 2001

U.S. Department of Energy



Lawrence
Livermore
National
Laboratory

DISCLAIMER

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

This work was performed under the auspices of the U. S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under Contract No. W-7405-Eng-48.

This report has been reproduced directly from the best available copy.

Available electronically at <http://www.doc.gov/bridge>

Available for a processing fee to U.S. Department of Energy
And its contractors in paper from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831-0062
Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-mail: reports@adonis.osti.gov

Available for the sale to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Road
Springfield, VA 22161
Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-mail: orders@ntis.fedworld.gov
Online ordering: <http://www.ntis.gov/ordering.htm>

OR

Lawrence Livermore National Laboratory
Technical Information Department's Digital Library
<http://www.llnl.gov/tid/Library.html>

Modern Tools for Modern Software

Gary Kumfert and Tom Epperly

October 31, 2001

Abstract

This is a proposal for a new software configure/build tool for building, maintaining, deploying, and installing software. At its completion, this new tool will replace current standard tool suites such as autoconf, automake, libtool, and the de facto standard build tool, make.

This ambitious project is born out of the realization that as scientific software has grown in size and complexity over the years, the difficulty of configuring and building software has increased as well. For high performance scientific software, additional complexities often arises from the need for portability to multiple platforms (including many one-of-a-kind platforms), multi-language implementations, use of third party libraries, and a need to adapt algorithms to the specific features of the hardware. Development of scientific software is being hampered by the quality of configuration and build tools commonly available. Inordinate amounts of time and expertise are required to develop and maintain the configure and build system for a moderately complex project. Better build and configure tools will increase developer productivity. This proposal is a first step in a process of shoring up the foundation upon which DoE software is created and used.

1 Introduction

This proposal is about scalability: not of software running on parallel machines, but of the tools commonly used to develop, deploy, and install the parallel software. In particular, this proposal focuses on how current configure and build tools do *not* adequately scale for today's scientific software. A configure tool automatically discovers the features of the hardware, operating system, and previously installed software that pertain to building a software package on that system. A build tool converts source material into end-use form. This typically includes translating source code to machine executable code (e.g. converting C or FORTRAN 77 into a program that can be run), installing files in a standard location, creating human readable documentation from source material, and other transformations needed to translate source material

into end-use form.

The current best practices in the community use 25 year old software development tools that have been incrementally adapted in an attempt to extend their usefulness. This process of incremental adaptation has produced something that can be made to work but requires considerable amounts of developer time to maintain. Under the sheer massiveness and complexity of today's scientific software, these tools are extremely unstable and fragile. Most importantly, this frailty is infused into the software it attempts to support.

Although there is inherent complexity in writing portable software, surprisingly little of this complexity is handled automatically by available software development tools. Instead, software portability is managed on a per project basis, demanding excessive time and expertise of each development team. As time and expertise vary from project to project, so does portability of software and adherence to current best practices.

The DoE has a significant investment in scientific software and rightly looks for ways to maximize the return on that investment. Component technology is an advanced software technique being explored to improve software reuse and interoperability in scientific computing — facilitating novel combinations of software. But this technology addresses interoperability only at the programming interface level. It does nothing to actually get software to build correctly on a particular piece of hardware under a particular operating system. Preliminary evidence suggests that component technology, by virtue of its assumption that software always installs trivially, actually raises the demands on an already strained software development infrastructure [10].

In addition to maintaining existing software, the DoE, also continually funds new efforts to produce new software. Ideally, the bulk of this funding would be spent on developing new algorithms or new capabilities, not on dealing with temperamental and fragile build tools. There is no known research that assigns a percentage value to how much DoE sponsored software development is spent on robustness and portability issues. Anecdotal evidence and personal experience seems to indicate that development overhead grows at least as the square of the number

of supported platforms.

Herein lies our argument in modern tools for modern software: If better tools can reduce developers' overhead for supporting additional platforms from quadratic to linear, or even quasi-linear, then there can be a dramatic increase in developer productivity, software can be substantially more portable, and a major stumbling block for component technologies would be removed; all with professional grade tools.

We believe that the time has come to stop incremental extensions to current tools, and start anew. There are new languages, abstractions, assumptions, features, and theory that can be built into new tools but could never be patched into exiting ones.

Section 2 discusses the tool that we feel is at the heart of our development scalability problem, a 25 year old tool called `make` [6]. It is reasonable to wonder if `make` is so old and so ineffective, why aren't other alternatives available? In fact, there are many alternatives, a sample of which are cataloged in Section 3. While putting to rest the argument that `make` can be vastly improved upon, this section raises a different question: with so many alternatives and no clear winner, what distinguishes our proposed effort? This question is addressed in Section 4; highlighting a strategy for accumulating user acceptance. We close in Section 5 by discussing big picture issues related to this project such as: how replacing `make` fits into a larger picture of software development, testing, deployment, installation, use and extension; how to gain user acceptance; what is the potential impact inside and outside the DoE.

2 Build Tools

Current state-of-art for building software in UNIX is to use a lowest common denominator solution: a primitive build tool developed in concert with the C language and the UNIX operating system in the 1970's called `make` [6]. Enticingly simple, most programmers learn to use it by copying examples, never needing to pick up a tutorial or reference book. It reads a file that enumerates target files, what files they depend on, and commands to "make" the target file. The commands are triggered whenever a dependent file has a more recent timestamp, which implies the target is out of date.

This build tool, `make`, doesn't know about directories, source code revision, compilers, installing software, programming languages, probing a new platform for its capabilities, software testing, debugging, bug tracking, or any other functions commonly associated with contemporary software development. These higher level functions are written into `make`'s rules — largely by hand — on a case by case basis. This effort is repeated in a non-standard

way for almost every software project written in C, C++, or F77 within the DoE.

Examining `make` only for the task that it was intended (refreshing stale files in the current directory) it still has significant deficiencies¹. `make` only triggers commands when it thinks files are out of date. It does not compensate for clock drift across a networked file system or for timestamping details when dealing with a version control system. In practice, developers can only be sure that everything is built correctly by removing all targets and starting from scratch. `make` has no mechanism for understanding that a command may produce more than one target file. To produce several targets, `make` will often execute the same command multiple times to satisfy its dependency rules. Perhaps most grievous is that the syntax of `make`'s input files (called Makefiles) is extremely error prone. One cannot visually inspect a Makefile for correctness, because whitespace (and even the type of whitespace) is significant. Using space characters (i.e. ASCII 32) instead of a tab character (i.e. ASCII 9) at the beginning of a line changes how `make` treats the line. Furthermore, trailing spaces at the end of a line can change the meaning of the subsequent line.

3 Previous Attempts to Unmake `make`

The current ubiquity of `make` is a fascinating example of the importance of being first to market, even with an inferior product. There have been, and still are, many attempts by Academia and the Open Source community to either improve `make`, or replace it outright.

The Free Software Foundation (FSF) arguably has the most feature-rich implementation of `make` called GNU-`make` [7]. Even so, they need layers of additional tools such as `autoconf`, `automake`, and `libtool` [16] to assist with configuration and building shared libraries. This strategy is one of the most successful and powerful in the Open Source community, but the proper orchestration of these independently developed and maintained tools requires an unreasonably high level of technical sophistication.

The X Windows² community has their own software build layer, `imake` [5, 8]. Other `make` variants include `nmake` [12] from Bell Labs, `dmake` [4], and `Open-Make` [13]. All of these have incremental enhancements

¹The lore surrounding `make` is that it was build by a summer intern in AT&T/Bell Labs over a week before taking a week's vacation. When he returned, so many people were using it that he was forbidden to change its syntax.

²X Windows is the windowing software in UNIX and predates Microsoft Windows

such as parallel execution of rules, more robust dependency checking, and other details, but still in keeping with the nuts-and-bolts lack of abstraction in make.

Open Source examples of tools that completely replace make—many also attempting a higher level of abstraction—are Ant [1] which is a Java-based build tool requiring developers to use XML, Cons [2] which is written in perl, Jam-Make/Redux [9] which is has a small, but loyal following, and Cook [3] which has been developed and maintained for about 15 years.

Despite all of these (and many other) efforts to enhance or outright replace make, none has succeeded in capturing the majority share of the market. There are two main reasons.

First, almost every UNIX or Linux system comes with make installed as a fundamental tool, and even non-UNIX environments support make to some extent. Thus software developers are confident that by using make they can reach the greatest number of users. Using any non-standard build tool becomes a significant perceived barrier to widespread user acceptance. Thus many make alternatives tend to be used in niche communities.

The second reason is specific to the problem domain of software build tools. There are thousands of ways to get it wrong. While there is widespread agreement that the current tools are bad, there is little agreement in what critical feature set is needed. An illustrative sample of these features is enumerated below³. Whenever we cite make as having a weakness, it infects all the tools that are built on top of make as well.

- **Handle spaces in strings gracefully.** Most tools including make require excessive (and careful) quoting of strings to preserve spaces within, Jam can't support them at all.
- **Cope with one action generating multiple files.** Most tools including make (and therefore everything built on top of make) can understand that a target file has multiple dependencies, but cannot fathom that a single action can produce several target files.
- **Cope with recursive builds.** Most software exists not as a single directory, but as a directory tree of hundreds, sometimes thousands of files. make has no built in concept of a directory. To compensate, the topmost Makefile invokes make itself in subdirectories. This admittedly gets the job done, but it is excessively slow [14, 17] and error prone [11].
- **Construct shared libraries.** This should be a simple task that is made complicated by the fact that there

³A separate document providing a more encyclopedic "critical feature set" is available separately from the same authors.

are no standards for how this is done. Each compiler vendor/operating system/hardware tuple has a different way of building a shared library, especially for C++. For a tool to properly address this problem alone would require an expert system with automatic update capabilities.

- **Manage partial development trees.** Typical large projects have individual coders or code-teams working on parts of the entire package. They would prefer to have a reference implementation of the entire package with a local copy of just their subtree, and build their subtree against the shared reference.
- **Build tool smarter than filesystem.** Most build tools (including make) does not have the sophistication to understand that two different Makefiles in two different directories can refer to the same file through different paths.
- **Flexible rule handling.** Build rules change on different platforms. make has no conditional branching to change its behavior on different platforms. This is accomplished in autoconf by actually creating makefiles from templates using simple text substitution.

Another notable effort to improve the current state of tools is the Software Carpentry Project [15] sponsored by CodeSourcery, LLC. and the Advanced Computing Laboratory at Los Alamos National Labs. This project funded a competition to design and implement open-source software tools that supersede current tools in four categories: configuration (e.g. autoconf), build (e.g. make), testing (e.g. XUnit, Expect, and dejaGNU) and defect tracking (e.g. gnats and bugzilla). This effort dictated that the tools be implemented in Python or Python-based systems, they adopt the project's Open Source license.

Despite initial high expectations, the Software Carpentry project enjoys only modest success. The winning designs for the testing and defect tracking tools are currently being developed by CodeSourcery, LLC. The build and configuration categories have both been abandoned after the second round of design competition. Unfortunately, both the scope and price tag of a solution were underestimated at the competition's inception.

4 Approach

We distinguish our effort from the others in several categories.

4.1 Establishing a Critical Feature Set

Establishing the "critical feature set" to make this project successful is will be the subject of much research, experi-

mentation, communication, and iteration. There is a substantial body of work already in print and working code on the subject. We also have access to all of the documents, reviews, and email from the Software Carpentry Project, as well as the support from the project's sponsors.

We will focus on developing a tool that is easier to use from the software developers' point of view. It will focus on day-to-day software development tasks, and be at a level of abstraction that is natural for developers. It should make the hard cases easier and keep the simple cases simple.

Under the hood, the tool will have more advanced algorithms and data structures to handle the finer control than currently available. It will have graph algorithms that handle a task having multiple dependencies or multiple results. It will allow rules to be modified by configuration results, results of tasks, or other rules. It will allow triggering of rules to be dynamically assigned from simple time-stamping, to context-sensitive file analysis. It will also allow the rules used to generate a target to depend on runtime details such as *"if machine load is low and I have 100MB of memory available, rebuild completely, otherwise just patch the critical piece."* It will be multi-threaded to handle parallel builds, and this multi-threadedness will be reflected in its syntax and not tacked on as an afterthought. It will also take the form of a core library to which various interfaces (e.g. XML, GUI, or native Python) can be attached.

4.2 Solve a Real Worst-Case Problem

First and foremost, we have a real software build problem. We have identified the Components project at Lawrence Livermore National Laboratory as an ideal candidate. This project has developed a tool called Babel that enables C, C++, F77, Python, and Java codes to interoperate. Unfortunately, Babel does nothing to support the build and installation of these interoperable pieces of software on various flavors of Solaris, Linux, OSF, AIX, and IRIX machines.

To configure and build the regression tests for this project, the developers of Babel are currently using GNU tools (GNUMake, autoconf, automake, and libtool), native (language specific) build functionality (Python and Java), and custom shell and perl scripts. Although their current solution works, it is completely unstable. Over the course of the last 1.5 years, their dominating cause of failure in the nightly builds was not the Babel software they are developing, but the improper orchestration of all of these build tools. Keeping the regression tests for this project running on all platforms (at 3700 tests per platform) has become a heroic developer effort, accounting for at least one FTE of the 5 FTE project.

The main concern of the Components Project, however, is that the solution for configuring and compiling this language interoperable regression suite cannot easily be reused by Babel's customers. Software developers who want to develop their own code using Babel will get the generated source code, but will have to struggle through their own build process. They do not (and indeed cannot) expect every code group who uses Babel to dedicate a person/year to learn the intricacies of GNUMake, automake, autoconf, libtool and all our helper scripts just to deploy and install language interoperable software.

We feel this is an ideal candidate to target our solution for. It has many desirable worst-case characteristics: multiple languages and multiple platforms. It already has explored some of the most advanced configure/build alternatives available and found them inadequate. It is also a project that we are both currently working on, so we already clearly understand the problems deeply and require no ramp-up time.

4.3 Getting User Buy-In Early and Often

DoE developers do not trust software that does not have a survival strategy... and they do not use what they do not trust.

Commercial software's survival hinges on the viability of the company and its ability to earn a consistent profit on maintaining the product. Survival of closed DoE software, depends on stable, long-term funding. Open-source software's survival strategy depends on garnering a "critical mass" of interested developer/users to share the burden of code development and maintenance.

The best survival strategy for this project is the open-source model. The reasons are legion. First and foremost; the dominant mode of distribution for scientific software is source code, whether it be open-source or licensed. Commercial build tools are plausible only when software is distributed in binary form; necessitating that the burden of software maintenance be solely on the original developers, and constraining the customer to a certain level of platform homogeneity.

The tools we are attempting to replace are open standards supported by Open Source tools. It is extremely unlikely that anything other than an Open Source tool could supplant make because make is already everywhere. One of the reasons people use make is because practically every development environment comes with a make compatible build tool. To replace make, a new build tool must become as common as make.

The biggest reason to court the favor of the OpenSource community is to earn the trust of DoE developers, and by earning their trust, we increase the potential for use. There are certain practices and policies that encourage the

OpenSource community's participation, but nothing can guarantee it. To maximize our chances of community participation in our project we propose the following actions.

- Licensing and copyright issues are showstoppers. These issues must be resolved at the beginning of the project. A standard Open Source or GNU license is far superior to a custom one. Developers are not lawyers. It is much easier for a developer to avoid software with an unfamiliar license, than to hire a lawyer to interpret it.
- Register the domain name `www.[foo].org`, where `[foo]` is the name of our (yet unnamed) tool. Maintain this website as a central place for all updates, news, and information regarding this tool. (See `www.python.org` for example.)
- Create an open repository on `sourceforge.org` and maintain all sources and documentation at that (remote) site. This allows anonymous users to download the current state of the software at all times, and it enables other developers to contribute to the software. This would require an exemption from the normal DoE Review and Release policies that require each line of code to go through Review and Release before being accessible off-site.
- Publish articles in trade journals (not scientific journals) to maximize number of software developers who have heard about the project and know where to find the website.
- Establish newsgroups and email forums for users and developers to discuss, argue, design, and learn. Provide web archives of all discussions at the aforementioned homepage.
- Encourage operating system vendors, operating system distributors and compiler vendors to include our build tool in their distribution.
- (future) Identify an existing Open Source project and convert its build system from make to our tool. One of the original authors of `autoconf` did this same exercise (he called it "autoconfiscating") with much success.
- (future) Publish a book about using our tool with an authoritative Open Source publishing house (e.g. O'Reilly and Associates, or New Riders Publishing)

4.4 Metrics for Success

The goal of this project is to increase programmer productivity. We propose replacing a vintage 1970's devel-

opment tool with a more powerful and sophisticated successor. We also predict that in doing so, we can contribute to more robust, portable, and interoperable software.

Measuring our success in increased programmer productivity is difficult and often requires duplication of effort in a controlled setting. Evidence to this effect, however can be reasonably inferred by tracking statistics such as: number of downloads, number of registered projects converting to our tool and abandoning make, number of contributions from the open-source community (fixes and extensions), and even setting up a computer outside our firewall to receive a network ping every 50th time a user actually uses our tool (with user's consent).

Another metric for success is how well we solve our real world, worst case problem identified in Section 4.2. If they can use our tools successfully, and if our tools make it easier for their customers to develop and deploy language-interoperable code, then we have demonstrated new functionality that is not in existing systems.

Publication is also a metric of success. Since this is more of a programmer tool than a scientific research project, we feel that trade journal publications should be weighted highly. Since we are advocating an open-source survival strategy for this project, we also think that trade journals should be weighted according to number of subscribers.

5 Summary

For this project, it's world domination or nothing. If all that comes out of this effort is yet another niche tool, we've lost.

The DoE is a pathological worst-case environment for software development, but the best place to develop — and perhaps in the most need of — our proposed tool. The DoE has some of the most sophisticated software in the world; also some of the longest lived, mathematically intense, and scientifically obscure. It has software written in a wide variety of languages, including custom languages. And this software runs on an even larger variety of platforms, including many custom and one-of-a-kind architectures.

As we continue developing higher-fidelity, multi-scale, multi-physics simulations, there is a continuing need to get all these pieces of software to fit together, often in ways that the original authors never intended. This activity stresses not only the software itself, but the tools that are used to build and maintain them. The breaking point, as evidenced by the Components Project, is clearly on the horizon.

Configuring and building software is a complicated, detailed, and unforgiving process. If we can raise the level at which developers articulate how their software should be

built, installed, and managed; if we can handle all of configuration, compilation, linking, and installation of software in one tool using one language; if we can convince developers that there is a true make alternative, just as universal, easier to use, less error prone, and more powerful; and if we can provide a mechanism where developers can collect large numbers of independently developed software libraries and bind them together into an massive parallel application with little or no difficulty, then we have a chance of making a contribution that can last for the next 25 years.

- [15] Software carpentry home-page. <http://software-carpentry.codesourcery.com>.
- [16] Gary V. Vaughan, Ben Elliston, Tom Tromey, and Ian Lance Taylor. *GNU Automake Autoconf and Libtool*. New Riders Publishing, October 2001.
- [17] Laura Wingerd and Christopher Seiwald. Constructing a large project with jam. In *Software Configuration Management SCM7*. International Conference on Software Engineering, May 1997.

References

- [1] Ant Website. jakarta.apache.org/ant.
- [2] Cons Website. www.dsmit.com/cons.
- [3] Cook Website. www.canb.auug.org.au/~millerp/cook/cook.html.
- [4] DMake Website. www.scri.fsu.edu/~dwyer/dmake.html.
- [5] Paul DuBois and Gigi Estabrook. *Software Portability with Imake*. Nutshell Handbooks. O'Reilly and Assoc., 1996.
- [6] Stuart I. Feldman. Make - a program for maintaining computer programs. *Software Practice and Experience*, 9(4):255-266, 1979. Revised April 1986.
- [7] GNUmake Website. www.gnu.org/manual/make-3.77/make.html.
- [8] IMake Website. www.prima.wisc.edu/software/imake-stuff/.
- [9] Jam-Make/Redux Website. www.perforce.com/jam/jam.html.
- [10] Gary Kurfert, Bill Bosl, Tamara Dahlgren, Tom Epperly, Scott Kohn, and Steve Smith. *Achieving Language Interoperability Using Babel*. CASC/ISCR Workshop on Component and Object-Oriented Technologies for Scientific Computing, Wente Vineyards, Livermore, CA, July 2001.
- [11] Peter Miller. Recursive make considered harmful. www.canb.auug.org.au/~millerp/rmch/recu-make-cons-harm.html, 1997.
- [12] NMake Product Builder. www.bell-labs.com/project/nmake.
- [13] OpenMake Website. www.catsyscorp.com.
- [14] Christopher Seiwald. Jam - make(1) redux. www.perforce.com/jam/doc/jam.paper.html, March 1994.